Introduction to Python

Seungwoo Schin

Coding the Mathematics Course, Fastcampus

I. 파이썬 소개

Listing 1: *Hello World! (hello.py)*

```
def main():
    print('Hello World!')

main()
```

본 단락에서는 파이썬 언어에 대해서 간단히 짚고 넘어갑니다.

I. 파이썬 데이터 타입

I.1 Built-in Data Types

파이썬은 기본적으로 다음의 빌트인 데이터 타입을 지원[1][2]합니다.

- Boolean Type : 참/거짓 값을 나타내는 타입입니다.
- Numeric Types : 일반적으로 쓰이는 숫자를 나타내는 타입입니다.
- Sequential Types : 배열 형태의 타입입니다.
- Mapping Types : key-value 순서쌍 형태의 타입입니다. [3]

아래에서는 각 타입의 종류와 다양한 연산법에 대해서 알아볼 것입니다.

Boolean Type 참(True), 거짓(False)값을 나타냅니다. Boolean 값들은 and, or, not 연산이 가능합니다. 연산의 결과는 아래와 같습니다.

Listing 2: *Boolean Example (example_bool1.py)*

```
assert (True and True) == True
assert (True and False) == False
assert (False and True) == False
assert (False and False) == False
assert (True or True) == True
assert (True or False) == True
assert (False or True) == True
assert (False or False) == False
assert (not True) == False
assert (not False) == True
```

[1] 참조 링크 1(위키북스), 참조 링크 2(공식 문서)
[2] 여기 리스트된 데이터형이 전부는 아니지만, 중요한 데이터형들이니 잘 알아두시길 권장합니다.
[3] 프로그래밍 지식이 있으신 분은 파이썬에서의 dict가 해쉬라고 생각하시면 좋습니다.

Numeric Types int, float, complex가 있습니다. 각각 정수, 실수[4], 그리고 복소수를 나타냅니다.

Listing 3: *Numeric Types (example_numeric1.py)*

```
a = 1
print(type(a)) # <type 'int'>
b = 1.0
print(type(b)) # <type 'float'>
c = 1.0 + 1j # j 알파벳은 그냥 변수지만 , 숫자 뒤에 붙어 오면 허수로
      인식합니다 .
print(type(c)) # <type 'complex'>
print('hello world!')
```

파이썬은 일반적인 사칙연산을 지원합니다. 아래에서 어떤 식으로 사칙연산이 사용되는지 볼 수 있습니다.

Listing 4: *Operations for Numeric Types (example_numeric2.py)*

```
print(1 + 2) # sum of x and y
print(3 - 1) # difference of x and y
print(2 * 4) # product of x and y
print(3 / 2) # quotient of x and y
print(3 // 2) # floored quotient of x and y
print(3 % 2) # remainder of x / y
print(-1) # x negated
print(+1) # x unchanged
print(abs(-2)) #  absolute value or magnitude of x
print(int(3.2)) # x converted to integer
print(float(2)) # x converted to floating point
```

Sequential Types 파이썬에서는 list, tuple, range, string 등의 배열 형태의 데이터형을 지원합니다. 여기서는 문자열 데이터형 string에 대해서 따로 다루지는 않으며, 더 자세한 정보는 공식 Documentation을 참고하시면 됩니다.

Listing 5: *Sequential Types (example_sequence1.py)*

```
a = [1,2,3,4] # list
b = (1,2,3,4) # tuple
c = range(10) # range
d = 'hello world!' # string
```

Sequence 데이터형들은 다음의 연산들을 지원합니다.

- a in d : 배열(d) 안에 특정 원소(a)가 있는지를 검사합니다.
- + : 배열 두 개를 이어서 새로운 배열을 만듭니다. 같은 데이터형이여야 합니다.
- d[i] : d의 i번째 원소를 반환합니다.
- d[i:j:k] : d의 i번째 원소부터 j번째 원소까지, k번째 원소마다 선택하여 리스트를 만들어 반환합니다.
- d.index(elem) : d에서 elem 이 처음으로 나오는 위치를 반환합니다.

[4]차후에 다루겠지만, 정확하게 실수를 나타내는 것은 불가능합니다. 더 정확하게는, 모든 실수를 정확하게 나타내는 것은 불가능합니다. 여기서의 float은 c언어에서의 double과 같다고 보는 것이 정확합니다.

Listing 6: *Operations for Sequential Types (example_sequence2.py)*

```
from example_sequence1 import *

print('e' in d) # True
print([1,2,3] + [4,5,6]) # [1,2,3,4,5,6]
print(d[1]) # 'e'
print(d[1:3]) # 'el'
print(d[1:6:2]) # 'el '
print(d[::-1]) # !dlrow olleh'
print(len(d)) # 12
for idx, elem in enumerate(d):
    print(idx, elem)
```

Mapping Types key-value 쌍을 저장하는 데이터형으로, 파이썬에서는 dict가 있습니다.

Listing 7: *Mapping Types (example_dict1.py, line 7-26 omitted)*

```
num2alphabet = \
    { 1 : 'a',
      2 : 'b',
      3 : 'c',
      4 : 'd',
      5 : 'e',
      26 : 'z', }
```

dict는 다음의 연산을 지원합니다.

- d[key] : dict에서 key에 해당되는 value를 반환합니다.
- d[key] = val : dict에서 key에 해당되는 value를 val로 업데이트합니다.
- d.keys() : dict의 key들을 반환합니다.

Listing 8: *Operations for Mapping Types (example_dict2.py)*

```
from example_dict1 import *

a[3] # 'c'
a.keys() # [1,2,3,...,26]
len(a) # 26
a[27]='A'
1 in a # True
del d[1]
1 in a # False
```

II. 파이썬 문법 : loops, conditionals

if-elif-else 다른 모든 언어와 비슷하게, 파이썬에서도 if-else 문을 지원합니다. 아래와 같은 문법으로 사용됩니다.

```
if cond1:
    # when cond1 is True
elif cond2:
```

```
    # when cond1 is False and cond2 is True
else:
    # when cond1 is False and cond2 is False
```

switch 파이썬에서는 switch문을 지원하지 않습니다. 하지만, 아래와 같이 switch문을 대체하여 사용할 수는 있습니다.

Listing 9: *Switch using dict (example_switch1.py)*

```
def func(a):
    switch_options = {\
        '1' : '1st',
        '2' : '2nd',
        '3' : '3rd', }
    return switch_options[a]
```

for loop 파이썬에서의 for loop는 임의의 Sequential Type 변수에 대해서, 그 변수 안의 원소를 한번씩 돌게 됩니다. 예를 들어서 아래 코드를 살펴봅시다.

Listing 10: *For Loop Example (example_for1.py)*

```
for elem in ['a', 'b', 'c']:
    print(elem)
```

while loop 파이썬에서의 while문은 다른 언어에서의 while문과 크게 다르지 않습니다. 아래의 소스코드를 살펴보면 알 수 있을 것입니다.

Listing 11: *While Loop Example (example_while1.py)*

```
i = 0
while i<10:
    print(i)
    i += 1
```

III. 파이썬 문법 : 함수, 클래스

III.1 함수

파이썬에서 함수는 다음과 같이 정의합니다.

Listing 12: *Function Syntax (example_function1.py)*

```
def function(args):
    return None
```

위 소스코드에서 각 항목은 아래와 같은 의미를 가집니다.

- def : 함수 정의 키워드입니다.
- function : 함수 이름을 나타냅니다.
- args : 함수 인자입니다. 아래와 같은 옵션이 있습니다.

  - arg
  - arg_default : 함수 인자의 기본값을 정해줄 때, =을 이용하여 기본값을 지정해줄 수 있습니다.
  - *arg_list : 정해지지 않은 수의 인자를 받고자 할 때, *을 하나 붙여서 들어온 인자를 배열로 받을 수 있습니다.
  - **arg_dict : 정해지지 않은 수의 이름이 명시된 인자를 받고자 할 때, *를 두개 붙여서 들어온 인자들을 dict 형태로 받을 수 있습니다.
- return None : 함수의 결과값으로 return 뒤의 구문을 반환합니다.

아래의 코드[5]를 보면 조금 더 명백해집니다.

Listing 13: *Function Argument Options (example_function2.py)*

```
def f(a = 0, *args, **kwargs):
    print("Received by f(a, *args, **kwargs)")
    print("=> f(a=%s, args=%s, kwargs=%s" % (a, args, kwargs))
    print("Calling g(10, 11, 12, *args, d = 13, e = 14, **kwargs)")
    g(10, 11, 12, *args, d = 13, e = 14, **kwargs)

def g(f, g = 0, *args, **kwargs):
    print("Received by g(f, g = 0, *args, **kwargs)")
    print("=> g(f=%s, g=%s, args=%s, kwargs=%s)" % (f, g, args, kwargs))

print("Calling f(1, 2, 3, 4, b = 5, c = 6)")
f(1, 2, 3, 4, b = 5, c = 6)
```

위 프로그램의 실행 결과는 다음과 같습니다.

Listing 14: *Output for Function Argmument Options*

```
Calling f(1, 2, 3, 4, b = 5, c = 6)
Received by f(a, *args, **kwargs)
=> f(a=1, args=(2, 3, 4), kwargs={'c': 6, 'b': 5}
Calling g(10, 11, 12, *args, d = 13, e = 14, **kwargs)
Received by g(f, g = 0, *args, **kwargs)
=> g(f=10, g=11, args=(12, 2, 3, 4), kwargs={'c': 6, 'b': 5, 'e': 14, 'd': 13})
```

람다 함수 람다 함수란 익명함수를 뜻합니다. 이는 람다함수가 변수명을 가질 수 없음을 의미하는 것이 **아닙니다.** 예컨대, 아래의 코드에서의 func1, func2는 둘 다 같은 함수(주어진 수에 2를 더하는)이며, func1은 람다식으로 작성되었지만 엄연히 func1이라는 이름을 가지고 있습니다.

Listing 15: *Lambda Function Example (example_lambda1.py)*

```
func1 = lambda x: x+2

def func2(x):
    return x+2

func3 = lambda x,y,z : x+y+z
func4 = lambda *args : sum(args)
```

[5] stackoverflow 질문 : Understanding kwargs in Python 참조

람다식의 문법은 위 소스 코드에서 볼 수 있듯이 다음과 같이 이루어집니다.

- lambda : 람다함수 키워드. 람다함수 뒤의 구문 중 콜론 전에 있는 구문은 함수의 인자를, 뒤는 반환하는 값을 나타낸다.
- x,y,z(func3)/*args(func4) : 람다함수의 인자. 쉼표로 구분되며, 상기된 *args등도 똑같이 사용가능함을 func4에서 확인할 수 있다.
- x+y+z(func3)/sum(args)(func4) : 람다함수의 반환값.

익명함수가 가지는 이점 중 하나는, 우리가 정수나 문자열을 다루듯이 함수 또한 하나의 변수로 다루고 싶을 때 편리하다는 점입니다. 본 단락에서는 일반화된 정렬 문제에서 어떤 식으로 람다식이 사용가능한지 보여드리고자 합니다.

어떤 배열을 정렬하는 문제를 생각해 봅시다. 이 때, 어떤 배열의 원소들이 정수라면 정렬 결과에는 이의가 없을 것입니다. 예컨대, 아래의 코드의 마지막 라인에서 AssertionError가 나지 않는다면 충분할 것입니다. [6]

Listing 16: *Inspecting Function Calls (example_lambda_sort1.py)*

```
def mysort(lst): # insertion sort
    if len(lst) == 1:
        return lst
    else:
        head, tail = lst[0], lst[1:]
        tail = mysort(tail)
        for idx, elem in enumerate(tail):
            if head <= elem:
                return tail[:idx] + [head] + tail[idx:]
        return tail + [head]

assert mysort([2,1,3]) == [1,2,3]
```

하지만 주어진 리스트가 비교하기 어려운 것들로 되어있는 경우 - 예를 들어서, 숫자 3개짜리 튜플로 되어있는 경우 - 에는 어떤 식으로 배열할 수 있을까요? 이를 위해서는 우선 배열의 원소를 서로 비교하기 위한 기준이 필요할 것입니다. 위 코드의 경우 원소간의 비교 기준은 대소관계이며, 8번째 라인(head>=elem)에 이것이 반영되었다고 볼 수 있습니다. 여기서는 이 기준을 세 숫자의 합으로 생각해 봅시다. 그렇다면, 새로운 기준(세 숫자의 합)을 아래와 같이 반영할 수 있을 것입니다.

Listing 17: *Inspecting Function Calls (example_lambda_sort2.py)*

```
def mysort(lst): # insertion sort
    if len(lst) == 1:
        return lst
    else:
        head, tail = lst[0], lst[1:]
        tail = mysort(tail)
        for idx, elem in enumerate(tail):
            if sum(head) >= sum(elem):
                return tail[:idx] + [head] + tail[idx:]
assert mysort([(1,2,3), (2,-4,2), (1,3,1)]) == [(1, 2, 3), (1, 3, 1), (2, -4, 2)]
```

이제 여기서 조금 더 나아가서, 다음과 같은 소스를 생각해 봅시다.

Listing 18: *Inspecting Function Calls (example_lambda_sort3.py)*

[6] 물론 실전에서는 더 많은 테스트를 하시는 것을 권장드립니다.

```
def compare(l, r): # (1)
    return sum(l) >= sum(r)

def mysort(lst, cmp): # (2)
    if len(lst) == 1:
        return lst
    else:
        head, tail = lst[0], lst[1:]
        tail = mysort(tail)
        for idx, elem in enumerate(tail):
            if cmp(head, elem): # (3)
                return tail[:idx] + [head] + tail[idx:]

assert mysort([(1,2,3), (2,-4,2), (1,3,1)],
              cmp = compare) == [(1, 2, 3), (1, 3, 1), (2, -4, 2)] # (4)
```

여기서, 위 소스를 조금 더 간소화해서 애초에 compare 함수를 def를 쓰지 않고 람다식을 이용하여 저렇게 쓸 수 있습니다.

Listing 19: *Inspecting Function Calls (example_lambda_sort4.py)*

```
def mysort(lst, cmp = lambda x,y: sum(x) >= sum(y)): # (2)
    if len(lst) == 1:
        return lst
    else:
        head, tail = lst[0], lst[1:]
        tail = mysort(tail)
        for idx, elem in enumerate(tail):
            if cmp(head, elem):
                return tail[:idx] + [head] + tail[idx:]

assert mysort([(1,2,3), (2,-4,2), (1,3,1)],
              cmp = compare) == [(1, 2, 3), (1, 3, 1), (2, -4, 2)] # (3)
```

III.2 클래스

본 단락에서는 파이썬에서 클래스를 어떻게 정의하는지를 살펴보고자 합니다.

클래스란 클래스는 흔히 과자틀과 그 과자틀로 찍어낸 과자로 비유됩니다. 클래스는 객체를 만들기 위한 코드 템플릿[7]을 말합니다. 인스턴스는 이 템플릿을 이용하여 만들어진 코드 덩어리로 생각할 수 있습니다. 예[8]를 들어서, 더하기만 가능한 간단한 계산기 코드를 만든다고 가정합시다. 다음과 같이 전역변수를 사용하면 어렵지 않게 구현할 수 있을 것입니다.

Listing 20: *더하기만 가능한 계산기 (example_cal1.py)*

```
result = 0

def adder(num):
    global result
```

[7] https://www.hackerearth.com/practice/python/object-oriented-programming/classes-and-objects-i/tutorial/

[8] 본 예시는 hrefhttps://wikidocs.net/28점프 투 파이썬에서 참고하였습니다.

```
    result += num
    return result

print(adder(3))
print(adder(4))
```

이 때, 각자 다른 계산기를 2개를 만들어서 사용하고자 하면, 다음과 같이 2개의 계산기를 만들어야 할 것입니다.

Listing 21: *더하기만 가능한 계산기 2개 (example_cal2.py)*

```
result1 = 0
result2 = 0

def adder1(num):
    global result1
    result1 += num
    return result1

def adder2(num):
    global result2
    result2 += num
    return result2

print(adder1(3))
print(adder1(4))
print(adder2(3))
print(adder2(7))
```

이렇게 코드를 작성할 경우, 완벽하게 똑같은 코드를 두 번 작성해야 함을 알 수 있습니다. 따라서 이러한 중복을 피하기 위해서, 파이썬에서는 - 그리고 객체지향적 언어에서는 - 클래스를 제공합니다. 클래스는 다음과 같이 계산기 코드를 템플릿화하여, 재사용이 용이하게 만듭니다.

Listing 22: *더하기 계산기 클래스 (example_cal3.py)*

```
class Calculator:
    def __init__(self):
        self.result = 0

    def adder(self, num):
        self.result += num
        return self.result

cal1 = Calculator()
cal2 = Calculator()

print(cal1.adder(3))
print(cal1.adder(4))
print(cal2.adder(3))
print(cal2.adder(7))
```

이 때, cal1, cal2는 클래스 Calculator의 인스턴스입니다.
클래스를 사용하는 이유는 개발의 속도와 유지보수의 편의성, 그리고 코드 디자인의 간결성 때문

입니다. 잘 구성된 클래스의 경우, 구현하고자 하는 대상과 구현하는 프로그램 소스 코드간 대응이 직관적입니다. 예컨대 회계사용 프로그램을 작성할 경우, 고객/법인/재무재표 등의 클래스를 사용하게 될 것입니다. 이런 경우, 단순 코드 블럭으로 되어있는 것보다 더 직관적으로 프로그램 전체의 구조를 이해할 수 있으며, 이는 개발 속도와 정확도에 긍정적인 영향을 끼칠 것입니다. 프로그램의 유지보수 측면에서도 이와 비슷한 이유로 추후 유지보수가 간결해집니다.

이제 조금 더 자세하게 클래스(특히, 파이썬에서의 클래스에 대해서) 알아보겠습니다.

파이썬에서의 클래스 위 단락에서 클래스가 무엇인지, 그리고 왜 필요한지에 대해서 간단하게 살펴보았습니다. 본 단락에서는 클래스를 구성하는 요소를 살펴보고, 이를 정의하는 파이썬 문법에 대해서 알아보고자 합니다. 이해를 돕기 위해서 여기서는 문자열을 다루는 간단한 클래스를 만들어보고자 합니다. 먼저,

클래스는 속성(attribute)와 메소드(method)로 구성됩니다. 속성은 클래스 속성과 인스턴스 속성으로 분류되며, 메소드는 인스턴스 메소드, 클래스 메소드, 스태틱 메소드로 분류됩니다. [9]

각각의 예시에 대해서 아래 코드에서 살펴보겠습니다. 먼저, 가장 기본적인 부분만을 작성한 코드를 살펴보겠습니다.

Listing 23: *Making a MyString Class (example_class1.py)*

```
class MyString:
    class_name = 'MyString'
    maker = 'Schin'

    datum = ''

a = MyString # a is a class
b = MyString() # b is an instance
```

위 코드에서 파이썬 클래스의 각 요소들을 설명하면 다음과 같습니다.

- class : 클래스를 정의하는 키워드입니다. 이 키워드 뒤의 단어가 클래스의 이름이 됩니다. 여기서는 MyString입니다.
- maker = 'Schin' : 클래스의 **속성**을 지정합니다. 여기서는 class_name, maker, datum의 3개의 속성이 있습니다. 편의상 datum을 우리가 다루고자 하는 문자열로 생각하겠습니다.
- a = MyString : a라는 변수의 값으로 MyString이라는 **클래스**를 지정합니다.
- b = MyString() : b라는 변수의 값으로 MyString이라는 **인스턴스**를 지정합니다.

여기서 클래스(a)와 인스턴스(b)의 차이를 살펴보기 위해서 다음 코드를 실행해 보겠습니다.

Listing 24: *Making a MyString Class (example_class1.py)*

```
print(a.class_name) # prints 'MyString'
print(b.class_name) # prints 'MyString'
a.class_name = 'new MyString'
print(a.class_name) # prints 'new MyString'
print(b.class_name) # prints 'new MyString'
b.class_name = 'MyString again'
print(a.class_name) # prints 'new MyString'
print(b.class_name) # prints 'MyString again'
```

[9] 파이썬의 경우, private method를 지원하기는 하나 완벽하게 private하지는 않습니다. __로 시작하는 속성이나 메소드는 private로 분류되지만, 이것이 완벽하게 클래스 밖에서 은닉되어 있지는 않습니다. 예컨대, 클래스 이름이 Mycls이고 속성이 __attr1일 경우, Mycls__attr1 로 접근할 수 있습니다. 더 자세한 내용은 참고 링크를 참고하세요.

여기서 볼 수 있듯이, 클래스 자체의 변화는 인스턴스에 영향을 주지만 그 역은 성립하지 않습니다. 이는 모든 인스턴스는 클래스의 틀을 따르기 때문입니다. 위에서의 쿠키틀의 예시를 들면, 쿠키 하나를 바꾼다고 쿠키틀이 바뀌지는 않는 것과 같습니다. 반대로, 쿠키틀을 바꾸면 모든 쿠키는 영향을 받게 됩니다.

본격적으로 문자열을 다루기 위해서 b에 우리가 다루고자 하는 문자열을 저장하고자 합니다. 이는 다음과 같은 방식으로 할 수 있습니다.

Listing 25: *MyClass Attribute (example_class1.py)*

```
b.datum = 'hello world!'
print(b.datum)
```

이제 이 후에는 어떻게 해야 할까요? 문자열을 저장하는 것만으로는 충분하지 않습니다. 이제 조금 더 복잡한 클래스 문법을 살펴보겠습니다.

Listing 26: *Making a MyString Class (example_class2.py)*

```
class MyString:
    class_name = 'MyString'
    maker = 'Schin'

    def __init__(self, datum = ''):
        self.datum = datum

    # instance method
    def get_first_letter(self):
        return self.datum[0]

    def most_frequent_word(self):
        bow = MyString._create_bow(self.datum)
        res = ''
        tmp = 0
        for k in bow.keys():
            if bow[k] > tmp:
                res = k
                tmp = bow[k]
        return res

    # magic method
    def __add__(self, other):
        if isinstance(other, self.__class__):
            return MyString(datum = self.datum + other.datum)
        return NotImplemented

    @classmethod
    def assign_author(cls, author):
        cls.author = author

    @staticmethod
    def _create_bow(input_str):
```

- 속성(Attribute) : 클래스나 인스턴스가 가지고 있는 정보를 말합니다. 예를 들어서, 문자열의 경우라면 문자열의 내용이나 저자 등이 있을 것입니다.

  - 클래스 속성 : 클래스 자체가 가지고 있는 속성을 말합니다. 예를 들어서, 문자열의 경우라면 클래스 이름 MyString 등이 있습니다.
  - 인스턴스 속성 : 인스턴스 각각이 가지고 있는 속성을 말합니다. 예를 들어서, 문자열의 경우라면 문자열의 내용 등이 있습니다.
- 메소드(method) : 클래스나 인스턴스의 상태를 변화시키거나, 새로운 클래스를 만드는 등의 작업을 하는 함수입니다.
  - 인스턴스 메소드 : 인스턴스에 대해서 작동하는 함수입니다. 인스턴스를 변형시키거나 인스턴스를 이용하여 어떠한 작업을 수행합니다.
  - 클래스 메소드 : 클래스에 대해서 작동하는 함수입니다. 클래스 자체를 변형시킬 수 있습니다.
  - 스태틱 메소드 : 클래스나 인스턴스와 상관없는 함수입니다. 보통 클래스 내에서만 쓰이지만, 딱히 인스턴스나 클래스의 정보를 필요로 하지 않는 경우에 쓰입니다.

위 소스 코드를 통해서 우리는 다음과 같은 일들을 할 수 있습니다.

Listing 27: *MyClass Attribute (example_class2.py)*

```
a = MyString('hello')
b = MyString('world')

print(a.get_first_letter())

c = a+b # MyString('helloworld')
print(c.datum)

c.assign_author('Shin')
print(c.author)
print(a.author)

d = MyString('hello world it is so nice to meet you how are you doing world')
print(d.most_frequent_word())
```

IV. 파이썬 인터프리터의 이해

본 단락에서는 주어진 소스 코드를 파이썬이 계산하는 법을 알아볼 것[10]입니다. 본격적인 설명에 앞서, 변수의 종류에 대해서 설명하겠습니다. 일반적으로 변수에는 다음의 세 종류가 있습니다.

- Bound variable : 어떤 값이나 다른 변수에 의해서 값이 결정되는지 정해진 변수
- Binding variable : Bound variable의 값을 결정하는 변수
- Free variable : Bound되지 않은 변수

이 때, 파이썬 인터프리터[11]는 **bound variable을 binding variable로 대체(substitute) 하여** 계산합니다. 만약 계산해야 하는 모든 변수들의 값이 결국 어떤 값(숫자, 문자열 등등)으로 환원되면 그 값을 계산하여 반환하고, 그렇지 않다면 에러를 반환합니다. 따라서 파이썬 인터프리터의 동작을 이해하는 것은 곧 어떤 식으로 변수들이 서로를 bind/bound 하는지를 이해하고, 이를 기반으로 **기계적으로** 변수를 적절한 값으로 대체하여 계산을 수행함을 의미합니다.

Binding이 일어나는 경우는 아래와 같습니다.

import문의 사용 import문을 사용할 경우, import된 모듈에서의 모든 namespace가 bind됩니다.

[10] 파이썬 공식 Documentation 참조
[11] 사실 많은 인터프리터가 대부분 이렇게 동작합니다.

for loop for loop에서 헤더는 루프 코드블럭 안에서 for loop 헤더에서 선언된 변수를 bind 합니다.

함수, 클래스의 정의 함수나 클래스의 정의는 함수나 클래스 이름을 bind하게 됩니다. 예를 들어서 아래 소스코드를 보면, compare이라는 변수는 1번 라인에 의해서 2번 라인에 binding되어, 14번 라인을 거쳐 11번 라인에서 쓰이게 됩니다.

Listing 28: *Binding in Function definition (example_lambda_sort3.py)*

```
def compare(l, r): # (1)
    return sum(l) >= sum(r)

def mysort(lst, cmp): # (2)
    if len(lst) == 1:
        return lst
    else:
        head, tail = lst[0], lst[1:]
        tail = mysort(tail)
        for idx, elem in enumerate(tail):
            if cmp(head, elem): # (3)
                return tail[:idx] + [head] + tail[idx:]

assert mysort([(1,2,3), (2,-4,2), (1,3,1)],
              cmp = compare) == [(1, 2, 3), (1, 3, 1), (2, -4, 2)] # (4)
```

=의 사용 예를 들어서, 아래의 소스코드에서는 각각 a,b가 bound variable, a가 binding variable, c가 free variable입니다.

Listing 29: *Types of Variables (variables.py)*

```
a = 1
b = a
c
```

함수 인자의 binding 함수 인자 역시 함수가 정의된 곳 내에서의 binding을 야기합니다. 예를 들어서 아래 소스코드를 살펴봅시다.

Listing 30: *Inspecting Function Calls (example_interpreter1.py)*

```
def func1(input_num):
    a = int(input_num[0])
    b = int(input_num[1])
    c = int(input_num[2])


    return func2(a,b,c,)

def func2(a,b,c):
    return 100*c + 10*b + a

print(func1('123'))
```

이 때 출력될 값은 321일 것입니다. 이를 이해하기 위해서 파이썬 인터프리터 안에서 어떤 일이 벌어지는지 한 단계씩 살펴보도록 하겠습니다.

1. line 12 : func1('123')을 호출, 반환된 값을 출력함
2. line 1 : func1 정의된 부분으로 감
3. line 2-7 : 이 부분의 식을 계산하되, input_num을 '123'으로 대체하여 계산함 (=binding이 일어남: input_num 이 '123'에 bind됨)
   (a) line 2-4 : int('123'[0]) == 1 과 같은 계산을 반복
   (b) line 7 : func2(1,2,3)을 호출, 반환된 값을 반환함
4. line 9 : func2 정의된 부분으로 감
5. line 10 : 100c + 10b + a 계산하되, a,b,c를 각각 1,2,3으로 대체하여 계산함 (=binding이 일어남 : a,b,c가 각각 1,2,3에 bind됨)